P02- Automatización de pruebas: Drivers

Os recordamos que el plazo para entregar los ejercicios de esta práctica (P02) termina justo ANTES de comenzar la siguiente sesión de prácticas (P03) en el laboratorio (los grupos de los lunes, por lo tanto, el próximo lunes, los de los martes, el próximo martes, ...). Pasado este plazo los profesores no revisarán vuestro trabajo sobre P02 (de forma individual), con independencia de que se indiquen en clase las soluciones que os permitirán saber si las vuestras son correctas.

Implementación de drivers con JUnit 5

En esta práctica implementaremos los drivers para automatizar la ejecución de los casos de prueba unitarios que hemos diseñado en la sesión anterior. Recuerda que a partir de ahora, nuestro elemento a probar se denominará SUT (con independencia de que sea una unidad o no), y por el momento, va a representar a una UNIDAD (que hemos definido como un método java).

Usaremos JUnit 5 para implementar nuestros tests, pero no se trata solamente de aprender el API de JUnit, sino de usarlo siguiendo las normas que hemos indicado en la clase de teoría: por ejemplo: los tests tienen que estar físicamente separados de las unidades a las que prueban, pero tienen que pertenecer al mismo paquete, los tests tienen que implementarse sin tener en cuenta el orden en el que se van a ejecutar, cada método anotado con @Test debe contener un único caso de prueba...

Ejecución de drivers con JUnit 5

La ejecución de los drivers se realizará integrada en el proceso de construcción del sistema, usando Maven. Es decir, de forma automática (pulsando un botón), se ejecutarán todas las actividades conducentes finalmente a obtener los informes de la ejecución de nuestros tests (casos de prueba). Es importante que tengas clara la secuencia de acciones de dicho proceso de construcción .

IntelliJ puede usar Maven, y también puede ejecutar los tests junit (sin usar maven) desde el menú contextual de cada fuente de test. IntelliJ controla los comandos maven, la versión usada, ... Pero nosotros queremos independizar nuestro proceso de construcción de cualquier IDE, de hecho hemos configurado IntelliJ para que no ejecute su propio Maven, sino el que hemos instalado en /usr/share de nuestra máquina virtual, de forma que si prescindimos del IDE y usamos Maven desde el terminal vamos a obtener siempre el mismo resultado. Esto no significa que no podáis ejecutar los tests directamente desde IntelliJ, pero recordad que tenéis que saber hacerlo a través de Maven. La *goal* ***surefire:test*** asociada por defecto a la fase test, será la encargada de ejecutar nuestros tests JUnit con Maven.

Bitbucket

El trabajo de esta sesión también debes subirlo a *Bitbucket*. Todo el trabajo de esta práctica deberá estar en el directorio **P02-Drivers**, dentro de tu espacio de trabajo, es decir, dentro de tu carpeta: ppss-2021-Gx-apellido1-apellido2

Recuerda que si trabajas desde los ordenadores del laboratorio primero deberás configurar git y clonar tu repositorio de Bitbucket.

Ejercicios

Vamos a crear un proyecto Maven, que contendrá todos los ejercicios de esta sesión. Para ello elegiremos la opción “File→New Project”. Seleccionaremos “Maven” en la parte izquierda de la siguiente pantalla, nos aseguraremos de que tiene asignado el JDK_11 y seleccionamos “Next”.

Primero proporcionaremos los valores para las coordenadas de nuestro proyecto, que serán “ppss” (para el GroupId), y “drivers” (para el ArtifactId). Usaremos la versión que aparece por defecto.

El campo de texto “Location” debe contener la ruta del directorio raíz de nuestro proyecto Maven. Por lo tanto asegúrate de que dicha ruta está dentro de tu directorio de trabajo, y en el subdirectorio P02-Drivers/drivers. Dado que la carpeta “drivers” no existe todavía, IntelliJ nos pedirá permiso para crearla (obviamente, le diremos que sí). La carpeta "drivers" creada será la carpeta raíz de tu proyecto Maven. Verás que el campo de texto "Name" tiene también el valor "drivers"

Pulsamos sobre "Finish" y puedes comprobar que en el subdirectorio P02-Drivers/drivers se ha creado un fichero pom.xml, el directorio src, y el directorio .idea (este último no tiene nada que ver con Maven, lo crea IntelliJ automáticamente en todos sus proyectos).

FICHERO POM.XML

Necesitamos modificar el fichero pom.xml. Para ello:

- Añade la propiedad *project.build.sourceEncoding,* con el valor *UTF-8* (recuerda que es obligatorio especificar esta propiedad).
- Las propiedades *maven.compiler.source* y *maven.compiler.target*, equivalen a la propiedad *maven.compiler.release* que ya hemos usado en la práctica anterior. Puedes sustituirlas por *maven.compiler.release*, o dejar las dos que aparecen por defecto.
- Necesitamos incluir el plugin *maven-compiler-plugin* para modificar la versión a la 3.8.0. Recuerda que por defecto se incluye una versión anterior con la que no podrás compilar usando jdk 11.
- Igualmente incluiremos el plugin *surefire* ("*maven-surefire-plugin*") para usar la versión 2.22.2.
- Añade la dependencia para usar junit (la tienes en las transparencias de teoría, recuerda que sólo necesitas usar la que hemos indicado en las transparencias de teoría).

Importante: Siempre debes modificar el pom.xml sin ayuda del IDE. Y debes tener claro para qué sirve cada elemento incluido en el mismo. Este fichero nos permite configurar la construcción del proyecto e incluir la automatización de nuestras pruebas en dicho proceso. La idea es que con sólo "pulsar un botón" (es decir, ejecutar UN comando maven) podamos compilar nuestros fuentes, compilar los tests, ejecutar los tests,...

La configuración que acabamos de hacer será la configuración básica que vamos a usar, a partir de ahora en todos los proyectos.

En las siguientes prácticas iremos añadiendo más elementos a partir de esta configuración básica, por lo que deberás recordarla.

➪ ➪ Ejercicio 1: *drivers* para `calculaTasaMatricula()`

Debes implementar y automatizar la ejecución de los casos de prueba que has diseñado en la sesión anterior para el método ***ppss.Matricula.calculaTasaMatricula().***

Nota: En caso de que no dispongas de la tabla de casos de prueba, puedes utilizar ésta:

	Datos de entrada			Resultado esperado
	edad	familia numerosa	repetidor	tasa
C1	19	falso	cierto	2000
C2	68	falso	cierto	250
C3	19	cierto	cierto	250
C4	19	falso	falso	500
C5	61	falso	falso	400

Tareas a realizar:

A) Necesitas el **código fuente de la unidad a probar**. Puedes copiarlo de las plantillas de la primera práctica. No olvides crear el **paquete ppss**. Asegúrate de que el código de tu SUT esté en tu disco duro en el directorio requerido por Maven.

SUT

B) **Implementa los drivers** asociados a tu tabla de diseño de casos de prueba (o usa la tabla que os hemos proporcionado). IntelliJ nos permite generar la clase de pruebas de la siguiente forma: muestra en el editor el código de la clase Matricula, selecciona el nombre de la clase, y desde el menú contextual pulsa sobre "Generate...→Test...". Asegúrate de que el test que se va a generar es JUnit5. Selecciona finalmente la casilla con la unidad a probar.

drivers

El nombre de cada driver será "C1_calculaTasaMatricula",... y así sucesivamente. Puedes usar las anotaciones que hemos visto en clase exceptuando @Tag y @ParameterizedTest..

Una vez implementados los tests, compílalos usando la fase **test-compile** de Maven. Para que IntelliJ nos muestre esa fase debes deseleccionar "Show Basic Phases Only" desde la rueda dentada de la ventana Maven Tienes que tener claro qué diferencia hay entre ejecutar *mvn compiler:testCompile* y *mvn test-compile.* Puedes comprobarlo si ejecutas previamente la fase clean antes de cada uno de dichos comandos.

mvn test compile

C) **Ejecuta los tests** desde la fase Maven correspondiente. Deberías ver en el informe Maven que se han ejecutado todos los tests. Para ver el informe de forma gráfica selecciona el icono con la "M" de color rojo a la derecha de la rueda dentada de la ventana Maven. En clase hemos explicado dónde se encuentran físicamente los fuentes, ejecutables e informes maven. Comprueba que verdaderamente es así.

mvn test

D) **Implementa** en una nueva clase de pruebas con nombre "**MatriculaParamTest**". Se trata de implementar un test parametrizado con los datos de la tabla de casos de prueba proporcionada. Deberás **modificar el pom** para añadir la dependencia correspondiente. **Nota**: todas las anotaciones que hemos visto en clase, que pueden usarse conjuntamente con la anotación @Test, se pueden usar igualmente con la anotación @ParameterizedTest.

test parametri-zado

Ejecuta de nuevo la fase test de Maven. Si has usado la misma tabla que para el apartado B) deberías obtener informes idénticos tanto para los drivers de MatriculaTest, como para los de MatriculaParamTest. Fíjate en que la goal surefire:test ejecuta TODOS los métodos anotados con @Test o @ParameterizedTest

E) **Sustituye** el caso de prueba C5 por (60, true, true, 400). Verás que detectamos un error. Depúralo.

➪ ➪ Ejercicio 2: drivers para buscarTramoLlanoMasLargo()

A partir de la tabla de casos de prueba que hayas diseñado en la sesión anterior para el método ***ppss.Llanos.buscarTramoLlanoMasLargo()***, se pide lo siguiente:

Nota: En caso de que no dispongas de la tabla de casos de prueba, puedes utilizar ésta:

TABLA A

	Datos de entrada	Resultado esperado
	lista de lecturas	Tramo
C1A	{3}	Tramo.origen =0 ; Tramo.longitud = 0
C2A	{100,100}	Tramo.origen = 0; Tramo.longitud = 1
C3A	{180, 180, 180}	Tramo.origen=0; Tramo.longitud=2

A) Añade el código fuente de la unidad en el **paquete ppss,** en la clase **Llanos** El código es el proporcionado en el enunciado de la práctica anterior (puedes copiarlo de la carpeta proporcionada **Plantillas-P02**). Añade también una nueva **clase Tramo**, con **dos atributos privados**: origen y longitud (de tipo entero), así como sus correspondientes **getters y setters**, más **dos constructores** para la clase: uno sin parámetros (en el que se inicializan a cero los dos atributos), y otro en el que los valores de los atributos se pasan como parámetros en el constructor. Para generar los constructores y los métodos puedes hacerlo desde el editor, y seleccionar desde el menú contextual "Generate....", seleccionar "Conctructor", y "Getter and Setter" respectivamente.

SUT

B) Implementa los **drivers** asociados a tu tabla de diseño de casos de prueba (o usa la tabla que os hemos proporcionado, TABLA A). La clase que contiene los drivers debe llamarse LlanosTest

drivers TABLA A

El nombre de cada driver será "**C1A_buscarTramoLlano**",... y así sucesivamente. No uses tests parametrizados

Nota: Para implementar los *drivers* debes tener en cuenta que el resultado obtenido es un OBJETO java. Por lo tanto, para comparar el resultado real con el esperado puedes: comprobar que cada uno de los valores de los campos son iguales, o bien redefinir el método equals() de la clase Tramo. Puedes probar a usar las dos opciones. El resultado debe ser idéntico. Para redefinir el método equals(), la forma más sencilla es hacerlo a partir del menú contextual del editor de IntelliJ seleccionando: ***"Generate...→equals() and hashCode()"***. Si haces las comprobaciones campo a campo recuerda que deberás agrupar las aserciones. Fíjate que es mucho mejor usar varias comprobaciones porque nos van a proporcionar más información en caso de fallo.

C) Añade a la clase LlanosTest, **tres nuevos drivers** asociados a la siguiente tabla adicional, obtenida también aplicando el método del camino básico. Los nombres de los drivers deben ser **C1B_buscarTramoLlano**,... y así sucesivamente.

drivers TABLA B

TABLA B

	Datos de entrada	Resultado esperado
	lista de lecturas	Tramo
C1B	{-1}	Tramo.origen =0 ; Tramo.longitud = 0
C2B	{-1, -1, -1, -1}	Tramo.origen = 0; Tramo.longitud = 3
C3B	{120, 140, -10, -10, -10}	Tramo.origen=2; Tramo.longitud=2

D) **Ejecuta todos los tests** y depura el código si detectas algún error. Recuerda que, aunque realices modificaciones en el código para depurarlo, TODOS los tests deben seguir "en verde". También debes intentar depurar el código SIN cambiar la estructura del CFG del método que queremos probar, si no lo haces así, tendrás que revisar la tabla, puesto que es probable que deje de ser efectiva y eficiente..

mvn test

E) **Implementa** en una nueva clase con nombre "LlanosParamTest" un **test parametrizado** con los datos de las tablas de casos de prueba A y B.

test parametri-zado

Ejecuta de nuevo la fase test de Maven, deberías obtener informes idénticos tanto para los drivers de LlanosTest, como para los de LlanosParamTest (si has usado las mismas tablas en ambos casos)

➪ ➪ Ejercicio 3: selección y filtrado de las ejecuciones de los tests

-Dgroups
-DexcludedGroups
-Dtest

Hemos visto en clase que la goal ***surefire:test*** contiene parámetros configurables, como por ejemplo las variables "**groups**" y "**excludedGroups**". Otro parámetro interesante (cuando se están desarrollando los drivers) es "**test**", que permite ejecutar una única clase y/o método. Podéis ver la lista completa de parámetros(variables) configurables en: https://maven.apache.org/surefire/maven-surefire-plugin/test-mojo.html).

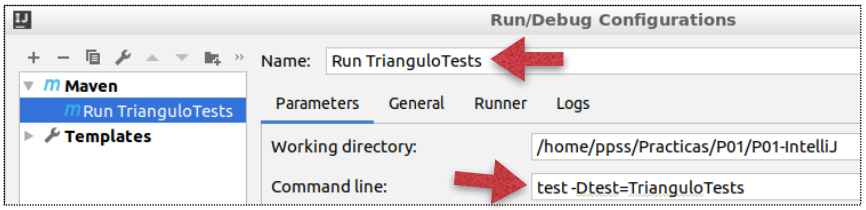
La variable test puede tener asignada una expresión regular, de forma que podamos seleccionar la ejecución de clases y/o métodos que sigan el patrón especificado, o puede usarse, que es como vamos a hacerlo nosotros, indicando clases y/o métodos concretos. Como ya hemos visto en clase, podemos cambiar la **configuración** del plugin desde el **pom**, o desde **línea de comandos**. La sintaxis desde línea de comandos es la siguiente:

```
mvn test –Dtest=ClaseAEjecutar1, ClaseAEjecutar2#metodoX, ...
```

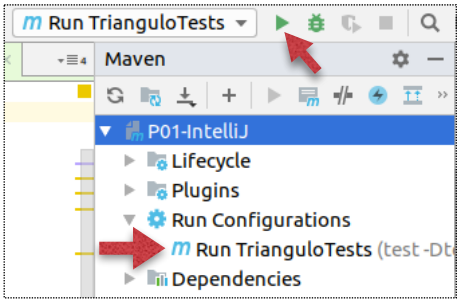
Desde IntelliJ podemos ejecutar cualquier comando Maven, pulsando sobre el icono correspondiente de la ventana Maven. No es necesario poner explícitamente el comando "mvn", sino la fase/s goal/s que queremos ejecutar, separadas por espacios.

Otra opción que nos resultará más útil, si por ejemplo queremos usar varios comandos mvn de forma frecuente es crearnos elementos "Configuration", de forma que podemos tener "guardados" diferentes comandos maven para poder usarlos cuando queramos, simplemente pulsando un botón.

Para crearnos una nueva "configuración", podemos hacerlo desde "*Run→Edit Configurations...*", o acceder directamente desde la barra de herramientas.



IntelliJ nos proporciona "plantillas" para poder usarlas con diferentes tipos de proyectos. Nosotros crearemos "Configurations" de tipo **Maven** (pulsando sobre el **icono +**). Simplemente tendremos que elegir un nombre e indicar el comando maven asociado.



Una vez creado, nos aparecerá como un nuevo "botón", anidado en el elemento "*Run Configurations*" de la ventana Maven. Podemos crear todas las "*Configurations*" que queramos, y también editarlas en cualquier momento.

También podremos ejecutar cualquier "*Configuration*" desde la barra de herramientas, eligiendo la que nos interese, y pulsando sobre el icono con forma de triángulo verde.

Nota: Cada vez que, desde el menú contextual del fichero java de un *driver*, ejecutamos los tests a través de IntelliJ, puedes comprobar que IntelliJ automáticamente crea una nueva "*Configuration*" para ejecutar los tests de esa clase, usando la plantilla correspondiente.

INSTRUCCIONES para GUARDAR LAS CONFIGURACIONES DE INTELLIJ (*Run Configurations)*

Intellij guarda todas las configuraciones que hemos creado en el fichero .idea/workspace.xml.

Por otro lado si muestras el contenido del fichero .gitignore (que se generó automáticamente al crear el repositorio), verás que aparece lo siguiente:

```
# JetBrains IDE
.idea/
```

es decir, que el directorio **`.idea`** y todo su contenido será ignorado por git, eso significa que no vamos a guardar en nuestro repositorio local ni en el remoto una copia del fichero workspace.xml.

Si desde otro ordenador desde el que habitualmene usas para hacer las prácticas, clonas tu repositorio remoto para seguir trabajando, no tendrás las *Run Configurations* que has creado en el otro ordenador, y por lo tanto tendrás que crearlas de nuevo si quieres usarlas.

Queremos ignorar los ficheros de ese directorio, pero estamos interesados en "conservar" las configuraciones, para poder seguir usándolas si nos vamos a otro ordenador y abrimos el proyecto con Intellij. Por lo tanto, lo que haremos será provocar que IntelliJ guarde dichas configuraciones en otro fichero "fuera" del directorio .idea.

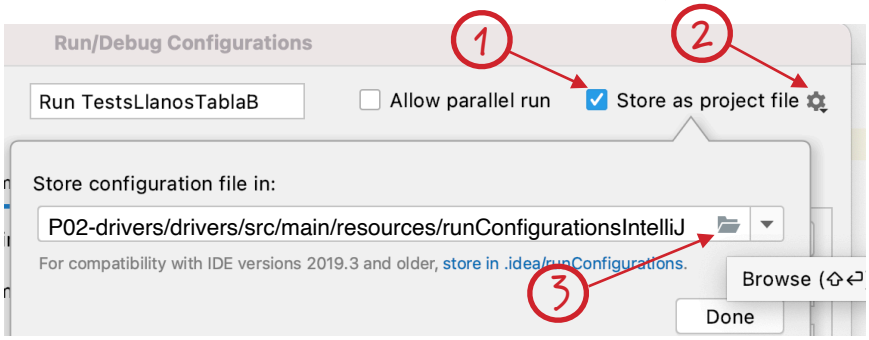
Concretamente las guardaremos en el directorio: `src/main/resources/runConfigurationsIntelliJ`.

El directorio **src/main/resources** es un directorio Maven, pensado para contener cualquier fichero adicional de nuestro proyecto que no sea código java (y que puede ser usado por nuestro código, o no, como en nuestro caso). Dentro de la carpeta src/main/resources podemos crear tantos subdirectorios como consideremos oportuno. Nosotros hemos decidido guardar las configuraciones en el subdirectorio ***runConfigurationsIntelliJ***

Para ello:

- En el directorio *src/main/resources*, crea la carpeta ***runConfigurationsIntelliJ.*** Puedes hacerlo dese la vista Projects, desde el menú contextual del elemento "Resources", accediendo a *New→Directory*

- Cuando añadas una nueva configuración debes marcar la casilla "***Store as project file***" e indicar la ruta donde guardaremos los ficheros asociadoc.



De esta forma, podremos guardar en Bitbucket dichos elementos, y seguir ignorando el directorio .idea.

Teniendo esto en cuenta, así como lo que hemos visto en clase, se pide:

A) Crea "Configurations" para ejecutar las clases con los drivers del Ejercicio1, y las del Ejercicio2, con los nombres "***Run ejercicio1***" y "***Run ejercicio2***", respectivamente (usando el parámetro "test" del plugin surefire). **Nota**: Si el valor que asignamos al parámetro "test" contiene espacios en blanco, deberás usar dobles comillas para que IntelliJ no lo interprete como el nombre de una fase y/o goal, es decir usaremos: -Dtest="clase1, clase2", en vez de. -Dtest=clase1, clase2

B) Etiqueta (anotación @Tag) el código de pruebas, de forma que podamos ejecutar todos los tests parametrizados, y crea una "configuration" con nombre "***Run Parametrizados***", o todos los tests sin parametrizar (y añade la "configuration" con nombre "***Run NO Parametrizados***"

C) Etiqueta (anotación @Tag) el código de pruebas, de forma que podamos ejecutar únicamente los tests de la Tabla A del ejercicio anterior, y crea una "configuration" con nombre "***Run TestsLlanosTablaA***", o sólo los tests de la Tabla B (y añade la "configuration" con nombre "***Run TestsLlanosTablaB***".

➪ ➪ Ejercicio 4: pruebas de excepciones y agrupaciones de aserciones

En el directorio **Plantillas-P02** encontrarás el fichero ***DataArray.java*** con una implementación para el método ppss.DataArray.delete(int).

La clase **DataArray** representa la tupla formada por una colección de datos enteros (hasta un máximo de 10) con valores mayores que cero, y el número de elementos almacenados actualmente en la colección. Los elementos ocupan siempre posiciones contiguas, y la primera posición será la 0. Las posiciones no ocupadas siempre tendrán el valor cero. La colección puede contener valores repetidos.

La especificación del método es la siguiente:

El método **delete(int)** borra el primer elemento de la colección cuyo valor coincida con el entero especificado como parámetro. El método lanzará una excepción de tipo DataException con un determinado mensaje, en los siguientes casos: cuando el elemento a borrar sea <=0 (mensaje: "El valor a borrar debe ser > cero"), cuando la colección esté vacía (mensaje: "No hay elementos en la colección"), cuando el elemento a borrar sea <= 0 y además la colección esté vacía (mensaje: "Colección vacía. Y el valor a borrar debe ser > cero"), y cuando el elemento a borrar no se encuentre en la colección (mensaje: "Elemento no encontrado").

Se proporciona la siguiente tabla de casos de prueba:

	Entradas		Resultado esperado
ID	colección, numElem	Elemento a borrar	(colección + numElem) o excepción de tipo DataException
F1	[1,3,5,7], 4	5	[1,3,7], 3
F2	[1,3,3,5,7], 5	3	[1,3,5,7], 4
F3	[1,2,3,4,5,6,7,8,9,10], 10	4	[1,2,3,5,6,7,8,9,10], 9
F4	[ ], 0	8	DataException(m1)
F5	[1,3,5,7], 4	-5	DataException(m2)
F6	[ ], 0	0	DataException(m3)
F7	[1,3,5,7], 4	8	DataException(m4)

m1: "No hay elementos en la colección"
m2: "El valor a borrar debe ser > cero"
m3:"Colección vacía. Y el valor a borrar debe ser > cero"
m4: "Elemento no encontrado"

Dada la especificación anterior del método *delete()*, implementa en una clase **DataArrayTest**, los *drivers* correspondientes a la tabla de casos de prueba proporcionada. Deberás agrupar las aserciones y comprobar que el mensaje de las excepciones generadas es el correcto.

Opcionalmente, puedes parametrizar los tests.

Añade una nueva "Run Configuration" para poder ejecutar únicamente la clase DataArrayTest, con el nombre "***Run DataArrayTest***" (no es necesario que uses la anotación @Tag). Recuerda que tienes que guardar la nueva configuración en el directorio ***runConfigurationsIntelliJ***, en la subcarpeta *resources* (del directorio *main*).

➪ ➪ ANEXO 1: Tabla de casos de prueba ejercicio "realizaReserva()"

En esta sesión no implementaremos (todavía) drivers para la tabla diseñada en el ejercicio 3 de la práctica anterior. Aunque implementaremos los drivers más adelante, dicha tabla es la siguiente:

	login	password	ident. socio	reserva()	isbns	Resultado esperado
C1	"xxxx"	"xxxx"	"Luis"	(1)	{"11111"}	??
C2	"ppss"	"ppss"	"Luis"	(2)	{"11111", "22222"}	No se lanza excep.
C3	"ppss"	"ppss"	"Luis"	(3)	{"33333"}	ReservaException2
C4	"ppss"	"ppss"	"Pepe"	(4)	["11111"}	ReservaException3
C5	"ppss"	"ppss"	"Luis"	(5)	{"11111"}	ReservaException4

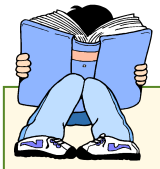
Suponemos que el login/password del bibliotecario es "ppss"/"ppss"; que "Luis" es un socio y "Pepe" no lo es; y que los isbn registrados en la base de datos son "11111", "22222".

ReservaException2: Excepción de tipo ReservaException con el mensaje: "ISBN invalido:33333; "
ReservaException3: Excepción de tipo ReservaException con el mensaje: "SOCIO invalido; "
ReservaException4: Excepción de tipo ReservaException con el mensaje: "CONEXION invalida;"
(1): No se invoca al método reserva()
(2): El método reserva() NO lanza ninguna excepción
(3): El método reserva() lanza la excepción IsbnInvalidoException
(4): El método reserva() lanza la excepción SocioInvalidoException
(5): El método reserva() lanza la excepción ConexiónInvalidaException

➪ ➪ ANEXO 2: Observaciones a tener en cuenta sobre la práctica P01B

- Nunca puede haber sentencias en el código que NO estén representadas en algún nodo
- Un nodo solo puede contener sentencias secuenciales, y NUNCA puede contener más de una condición
- El grafo tendrá un único nodo inicio y un único nodo final
- Todas las sentencias de control tienen un inicio y un final, debes representarlos en el grafo.
- Todos los caminos independientes obtenidos tienen que ser posibles de recorrer con algún dato de entrada,
- Cada caso de prueba debe recorrer "exactamente" todos los nodos y aristas de cada camino
- No podemos obtener más casos de prueba que caminos independientes hayamos obtenido
- Todos los datos de entrada deben de ser concretos
- Posiblemente necesitaremos hacer asunciones sobre los datos de entrada (como en la tabla mostrada en el anexo 1).

Resumen



¿Qué conceptos y cuestiones me deben quedar CLAROS después de hacer la práctica?

IMPLEMEMENTACIÓN DE LOS TESTS

- Es necesario haber diseñado previamente los casos de prueba para poder implementar los drivers.
- El código de los drivers estará en src/test/java, en el mismo paquete que el código a probar. Nuestra SUT será una unidad, por lo tanto, estaremos automatizando pruebas unitarias, usando técnicas dinámicas. Tendremos UN driver para CADA caso de prueba.
- Debemos cuidar la implementación de los tests, para no introducir errores en los mismos.
- Para compilar los drivers "dependemos" de la libreria JUnit 5, que habrá que incluir en el pom. Antes de compilar el código de los drivers, es imprescindible que se hayan generado con éxito los .class del código a probar (de src/main/java). Es decir, nunca haremos pruebas sobre un código que no compile.
- Debes usar correctamente tanto las anotaciones Junit (@) como las sentencias explicadas en clase (Assertions)

EJECUCIÓN DE LOS TESTS

- La ejecución de los tests se integra en el proceso de construcción del sistema, a través de MAVEN, (es muy importante tener claro que "acciones" deben llevarse a cabo y en qué orden).. La goal surefire:test se encargará de invocar a la librería JUnit en la fase "test" para ejecutar los drivers.
- Podemos ser "selectivos" a la hora de ejecutar los drivers, aprovechando la capacidad de Junit5 de etiquetar los tests..
- En cualquier caso, el resultado de la ejecución de los tests siempre será un informe que ponga de manifiesto las discrepancias entre el resultado esperado (especificado), y el real (implementado). Junit nos proporciona un informe con 3 valores posibles para cada test: pass, fault y error.
- Si durante la ejecución de los tests, alguno de ellos falla, el proceso de construcción se DETIENE y termina con un BUILD FAILURE.
- Debes tener claro que comandos Maven debemos usar y qué ficheros genera nuestro proceso de construcción en cada caso.